Introduction

This paper discusses three different approaches, and several variations on them, to solving the problem of scheduling non-identical jobs on a batch processing machine. Batch processing machines, for the purposes of this paper, can process multiple, non-identical jobs simultaneously – but all jobs must be loaded into and unloaded from the machine at once, which introduces a considerable twist on the "simple parallel resources" known from typical example problems in existing literature.

The machines in question represents real-life resources like autoclaves or ovens, which can process multiple items at a time, but often cannot be opened at random – in fact, such machines often need to wait for the largest item in the batch to be done before the next batch can be inserted.

Malapert proposed a custom constraint programming algorithm consisting of a set of filtering rules to solve the problem. He achieved considerably better speeds than with a simple mixed-integer model. In this paper, we present 1) an improvement to his MIP model, 2) a CP model and 3) a decomposition approach to "divide and conquer" the problem.

1.1 Problem definition

We describe the problem as follows: assume we are given a set of jobs J, each of which has a processing time (or "length") p_j and a size (or "capacity requirement") s_j – this notation follows Malapert's paper. Each job also has a due date d_j. The machine is characterized by its capacity b, and in every batch, the jobs' summed sizes must not exceed this number. All values are integer.

The machine can only process one batch k of jobs at a time, and batches always take as long as the longest job in the batch (i.e. $P_k = \max_{j \in k}(p_j)$). Our objective is to minimize the lateness L of the latest job in J, where L is the difference between

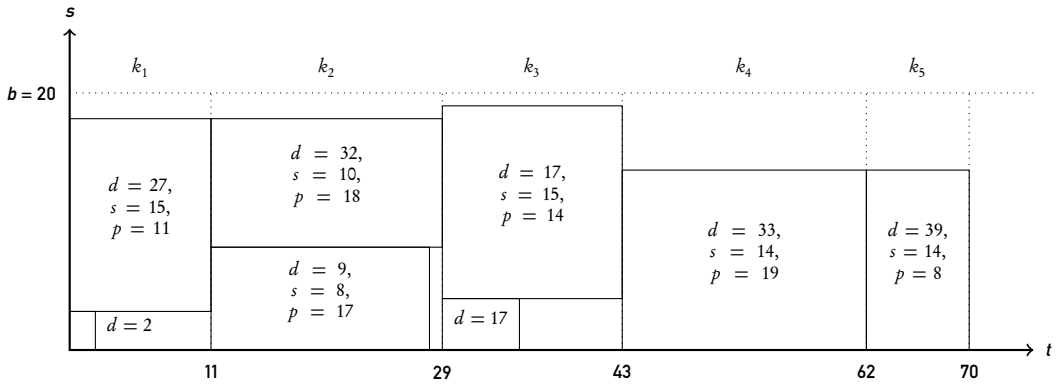


Figure 1.1: Optimal solution to an example problem with eight jobs (s_j and p_j not shown for the two small jobs)

the job's completion time C_j and its due date d_j – in formal terms, *min.* $L_{\max} = \max_j(C_j - d_j)$. The job's completion time, however, is the completion time of the batch, which in turn finishes with its *longest* job as stated above.

Malapert uses the format established by Graham et al. [?] to summarize the problem as $1|p\text{-}batch; b < n; non\text{-}identical|L_{\max}$, where $p\text{-}batch; b < n$ represents the parallel-batch nature and the finite capacity of the resource. A simpler version with identical job sizes was shown to be strongly NP-hard by Brucker et al. [?], and this problem, then, is no less difficult.

It helps to visualize the jobs before delving into the technicalities of scheduling them. Figure 1.1 shows a solution to a sample problem with eight jobs and a resource with capacity $b = 20$.

1.2 Organization of this paper

After reviewing some of the most relevant publications on both general MIP/CP models and batch scheduling problems, we first describe Malapert's original MIP model in section 3.1. We then present possible improvements to the model in 3.2. Section 3.3 introduces a CP formulation of the same problem. Sections 3.4.1 and 3.4.1 describe a decomposition approach.

An empirical comparison of the methods and a discussion of the results follow in sections 4.1 and 5. Ideas for future work are listed in 6.

Literature Review

Papers I have read so far:

2.1 General MIP and CP

- Fundamentals of mathematical programming
- Benders

2.2 Batch processing

- Azizoglu
- Dupont
- Sabouni

2.2.1 MIP

- Grossmann

2.2.2 CP

- Baptiste
- Malapert (how does he fit into the picture?)

Modelling the problem

3.1 MIP model

Malapert's original MIP approach, as given in Model 3.1.1, uses a set of binary decision variables x_{jk} to represent whether job j is assigned to batch k. The original model assumes a set K of $|K| = |J|$ batches, the number of jobs being a trivial upper bound on the number of batches required; it also enforces an earliest-due-date-first (EDD) ordering of the batches (constraint 3.7).

$$\begin{aligned}
\text{Min.} \quad & L_{\max} && && (3.1)\\
\text{s.t.} \quad & \sum_{k \in K} x_{jk} = 1 && \forall j \in J && (3.2)\\
& \sum_{j \in J} s_j x_{jk} \leq b && \forall k \in K && (3.3)\\
& p_j x_{jk} \leq P_k && \forall j \in J, \forall k \in K && (3.4)\\
& C_{k-1} + P_k = C_k && \forall k \in K && (3.5)\\
& (d_{max} - d_j)(1 - x_{jk}) + d_j \geq D_k && \forall j \in J, \forall k \in K && (3.6)\\
& D_{k-1} \leq D_k && \forall k \in K && (3.7)\\
& C_k - D_k \leq L_{\max} && \forall k \in K && (3.8)\\
& C_k \geq 0, P_k \geq 0 \text{ and } D_k \geq 0 && \forall k \in K && (3.9)
\end{aligned}$$

Model 3.1.1: Malapert's original MIP model

3.2 Improved MIP model

Several improvements can be made to Malapert's MIP model in the form of additional constraints shown in Model 3.2.1, as described in greater detail in the subsections below.

$$\sum_{j\in J} x_{j,k-1} = 0 \rightarrow \sum_{j\in J} x_{jk} = 0 \quad \forall k \in K \tag{3.10}$$

$$e_k + \sum_{j\in J} x_{jk} \geq 1 \quad \forall k \in K \tag{3.11}$$

$$n_j(e_k - 1) + \sum_{j\in J} x_{jk} \leq 0 \quad \forall k \in K \tag{3.12}$$

$$e_k - e_{k-1} \geq 0 \quad \forall k \in K \tag{3.13}$$

$$x_{jk} = 0 \quad \forall \{j \in J, k \in K | j > k\} \tag{3.14}$$

$$L_{\max} \geq \lceil \frac{1}{b} \sum_j s_j p_j \rceil - \delta_q \quad \forall q, \forall \{j \in J | d_j \leq \delta_q\} \tag{3.15}$$

Model 3.2.1: Improvements to Malapert's original MIP model

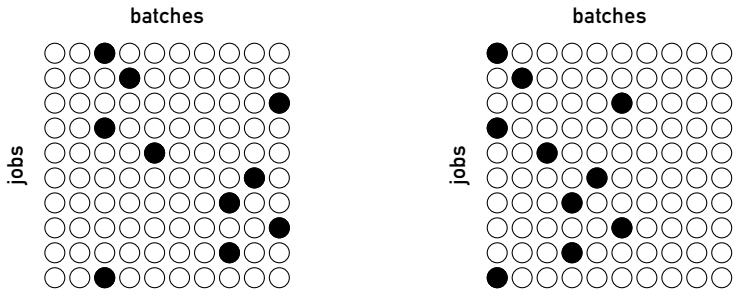
3.2.1 Grouping empty batches

The given formulation lacks a rule that ensures that no empty batch is followed by a non-empty batch. Empty batches have no processing time and a due date only bounded by $d_{\max}$, so they can be sequenced between non-empty batches without negatively affecting $L_{\max}$. Since, however, desirable schedules have no empty batches scattered throughout, we can easily reduce the search space by disallowing such arrangements. The idea is illustrated in Figure 3.1.

A mathematical formulation is

$$\sum_{j\in J} x_{j,k-1} = 0 \rightarrow \sum_{j\in J} x_{jk} = 0 \quad \forall k \in K. \tag{3.16}$$

To implement this, we can write constraints in terms of an additional set of binary variables, e_k, indicating whether a batch k is empty or not:



(a) Without dominance rule (b) With dominance rule

Figure 3.1: Dominance rule to eliminate empty batches followed by non-empty batches (circles represent the x_{jk} variables; a filled circle stands for $x_{jk} = 1$)

$$e_k + \sum_{j \in J} x_{jk} \geq 1 \qquad \forall k \in K, \tag{3.17}$$

$$n_j(e_k - 1) + \sum_{j \in J} x_{jk} \leq 0 \quad \forall k \in K. \tag{3.18}$$

Constraints (3.17) enforce $e_k = 1$ when the batch k is empty. Constraints (3.18) enforce $e_k = 0$ otherwise, since the sum term will never exceed n_j. The rule (3.16) can now be expressed as $e_{k-1} = 1 \rightarrow e_k = 1$, and implemented as follows:

$$e_k - e_{k-1} \geq 0 \quad \forall k \in K. \tag{3.19}$$

We can also prune any attempts to leave the first batch empty by adding a constraint $e_0 = 0$.

3.2.2 No postponing of jobs to later batches

Since the jobs are already sorted by non-decreasing due dates, it makes sense to explicitly instruct the solver never to attempt to push jobs into batches with a greater index than their own: even if every job had its own batch, it would be unreasonable to ever postpone a job to a later batch.

$$x_{jk} = 0 \quad \forall \{j \in J, k \in K | j > k\} \tag{3.20}$$

3.2.3 Lower bound on $L_{\max}$

Let a *bucket* q denote the set of all batches with due date δ_q. Then the completion date C_q of this bucket is the completion date of the last-scheduled batch with due date δ_q, and the lateness of the bucket q is $L_q = C_q - \delta_q$. Since all batches up to and including those in bucket q are guaranteed to contain all jobs with due dates $d \leq \delta_q$ – as ensured by the EDD ordering of batches – the lower bound on every bucket's lateness $LB(L_q)$ is a valid lower bound on $L_{\max}$. In other words, jobs with due date $d \leq \delta_q$ will be found only in batches up to and including the last batch of bucket q. This provides a lower bound on the lateness of bucket q:

$$L_{\max} \geq C_{\max,q} - \delta_q \quad \forall q \tag{3.21}$$

The buckets up to bucket q will likely also contain some later ($d > \delta_q$) jobs in the optimal solution but this does not affect the validity of the lower bound.

Now we need to find $C_{\max,q}$, or at least a lower bound on it, in polynomial time. The simplest approach simply considers the jobs' total "area" (or "energy"), i.e. the sum of all $s_j p_j$ products:

$$C_{\max,q} \geq \left\lceil \frac{1}{b} \sum_j s_j p_j \right\rceil \quad \forall q, \forall \{j \in J | d_j \leq \delta_q\} \tag{3.22}$$

A better lower bound on $C_{\max,q}$ would be given by a preemptive-cumulative schedule. Unfortunately, minimizing $C_{\max}$ for such problems is equivalent to solving a standard bin-packing problem, which requires exponential time.[1]

3.3 CP model

The constraint programming model is based on a set of decision variables $B_j = \{k_1, \ldots, k_{n_k}\}$, where each variable stands for the batch job j is assigned to. The complete model is given in Model 3.3.1.

Constraint (3.24) makes sure the jobs are distributed into the batches such that no batch exceeds the capacity b. Constraint (3.25), a global cumulative constraint, keeps jobs from overlapping on the resource – while redundant with (3.24), this speeds up propagation slightly. Constraints (3.26), (3.27), (3.28) and (3.29) define P_k, D_k, C_k (batch completion time) and $L_{\max}$, respectively.

[1] In a preemptive-cumulative schedule, jobs may be stopped and restarted mid-execution, but occupy a constant amount s_j on the resource while executing. In such a schedule, minimizing the makespan is as difficult as solving a bin-packing problem: we can break jobs into small pieces (no longer than the smallest common divisor of the jobs' lengths p) and then pack them together such as to minimize the number of small time slots needed.

$$\begin{aligned}
\text{Min.} \quad & L_{\max} && && (3.23)\\
\text{s.t.} \quad & \texttt{pack}(J,K,b) && && (3.24)\\
& \texttt{cumul}(J,b) && && (3.25)\\
& P_k = \max_j p_j && \forall\{j \in J | B_j = k\}, \forall k \in K && (3.26)\\
& D_k = \min_j d_j && \forall\{j \in J | B_j = k\}, \forall k \in K && (3.27)\\
& C_k + P_{k+1} = C_{k+1} && \forall k \in K && (3.28)\\
& L_{\max} \geq \max_k (C_k - D_k) && && (3.29)\\
& \texttt{IfThen}(P_k = 0, P_{k+1} = 0) && \forall k \in \{k_1, \ldots, k_{n_k - 2}\} && (3.30)\\
& B_j \leq k && \forall\{j \in J, k \in K | j > k\} && (3.31)
\end{aligned}$$

Model 3.3.1: Constraint programming model

3.3.1 Grouping empty batches

We can force empty batches to the back and thus establish dominance of certain solutions. The implementation is much easier than in the MIP model:

$$\texttt{IfThen}(P_k = 0, P_{k+1} = 0) \quad \forall k \in \{k_1, \ldots, k_{n_k - 1}\} \tag{3.32}$$

3.3.2 No postponing of jobs to later batches

Just like in the MIP model, jobs should never go into a batch with an index greater than their own:

$$B_j \leq k \quad \forall\{j \in J, k \in K | j > k\} \tag{3.33}$$

3.4 Decomposition approach

Instead of solving the entire problem using one model, we can solve the problem step by step, using the best techiques available for each subproblem. This approach is inspired by a method called *Benders decomposition*.

3.4.1 Branch-and-bound by batch in chronological order

A basic version of this approach uses branch-and-bound to transverse the search tree. At each node on level ℓ, a single MIP and/or CP model is run to assign jobs to batch $k = \ell$. The remaining jobs are passed to the children nodes, which assign jobs to the next batch, and so on – until a solution, and thus a new upper bound on $L_{\max}$, is found. Several constraints are used to prune parts of the search tree that are known to offer only solutions worse than this upper bound. Figure 3.2 shows an example in which a MIP model is used to assign jobs to the batch at every node. Algorithm 1 outlines what happens at each node: the model finds the best jobs to assign to the batch according to some rule, lets the children handle the remaining jobs, and tries the next best solution once the first child has explored its subtree and backtracked.

Algorithm 1 MIP node class code overview

```
update currentAssignments                          ▷ this keeps track of where we are in the tree
if no jobs given to this node then                 ▷ if this is a leaf node, i.e. all jobs are assigned to batches
    calculate L_max,current based on currentAssignments
    if L_max,current < L_max,incumbent then
        L_max,incumbent ← L_max,current
        bestAssignments ← currentAssignments
    end if
    return to parent node
end if
set up MIP model                                   ▷ as described below
repeat
    x_j ← model.solve(x_j)                         ▷ let model assign jobs to the batch
    spawn and run child node with all {j|x_j = 0}  ▷ pass unassigned jobs to children
    add constraint to keep this solution from recurring  ▷ this happens once the child node returns
until model has no more solutions
return to parent node
```

Using MIP and cumulative packing after the batch

The first version of the branch-and-bound batch-by-batch decomposition uses a MIP model at each node to assign jobs to the respective batch. The remaining jobs are packed such as to minimize their $L_{\max}$, with a relaxation of the batching requirement, i.e., as if on a cumulative resource.

Model 3.4.1 implements a time-indexed cumulative constraint on the non-batched jobs. Constraints (3.35) through (3.37) ensure that the batch stays below capacity, define the duration of the batch P_k and force at least one of the earliest-due jobs into the batch.

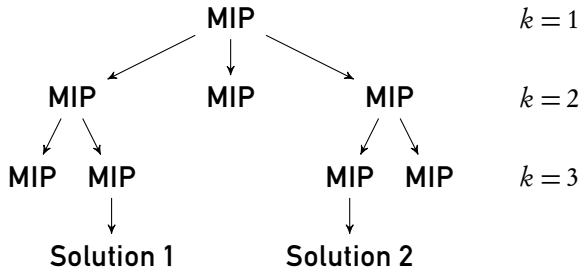


Figure 3.2: Batch-by-batch decomposition using MIP only

x_j	is 1 iff job j is assigned to the batch
u_{jt}	is 1 iff job j starts in time slot t
T_{jt}	is the set of all time slots occupied by job j if it ended at time t, that is $T_{jt} = \{t - p_j + 1, \ldots, t\}$
v_k	is the start time of the batch at the given node in the search tree
$L_{\text{max,incmb}}$	is the incumbent (known best) value of and thus an upper bound on $L_{\max}$
$\mathcal{H}$	is the set of all indexed time points $\{0, \ldots, n_t - 1\}$
w_j	is 1 iff job j is either longer than the batch ($p_j > P_k$) or already part of the batch ($x_j = 1$). Neither condition must be fulfilled for constraint (3.45) to have an effect on job j

Table 3.1: Notation used in the decomposition model

Constraints (3.38) express the interval relaxation used to ensure that no jobs exceed their latest allowable finish date given any batch assignment. Even jobs that are assigned to the batch have to fulfill this requirement.

Constraints (3.39) and (3.40) implement the cumulative nature of the post-batch assignments by ensuring that each job starts only once, and no jobs overlap on a given resource at any time.

Constraints (3.41) again limit the possible end dates of a job, but unlike (3.38), they use the time assignments on the cumulative resource to determine end dates. Constraints (3.42) define the value of $L_{\text{max,cumul}}$, the maximum lateness of any job in the non-batched set.

Constraints (3.43) force batched jobs to start at $t = 0$, while (3.44) force *all* jobs to start either at $t = 0$ or after the last batched job ends.

Constraints (3.45) enforce a dominance rule: jobs must be assigned to the batch

$$\text{Min.} \quad L_{\max,\text{cumul}} \tag{3.34}$$

$$\text{s. t.} \quad \sum_j s_j x_j \leq b \qquad \forall j \in J \tag{3.35}$$

$$P_k \geq p_j x_j \qquad \forall j \in J \tag{3.36}$$

$$\sum_j x_j \geq 1 \qquad \forall \{j \in J | d_j = \min(d_j)\} \tag{3.37}$$

$$P_k + \frac{1}{b}\sum_i s_i p_i \leq d_j + L_{\max,\text{incmb}} - 1 - v_k \qquad \forall j \in J, \forall \{i \in J | d_i \leq d_j\} \tag{3.38}$$

$$\sum_t u_{jt} = 1 \qquad \forall j \in J \tag{3.39}$$

$$\sum_j \sum_{t' \in T_{jt}} s_j u_{jt'} \leq b \qquad \forall t \in \mathscr{H} \tag{3.40}$$

$$(v_k + t + p_j) u_{jt} \leq d_j + L_{\max,\text{incmb}} - 1 \qquad \forall j \in J, \forall t \in \mathscr{H} \tag{3.41}$$

$$L_{\max,\text{cumul}} \geq (v_k + t + p_j) u_{jt} - d_j \qquad \forall j \in J, \forall t \in \mathscr{H} \tag{3.42}$$

$$u_{j,t=0} = x_j \qquad \forall j \in J \tag{3.43}$$

$$u_{it} \leq (1 - x_j) \qquad \forall i, j \in J, \forall t \in \{1, \ldots, p_j - 1\} \tag{3.44}$$

$$b - \sum_{i \in J} s_i x_i \leq (b w_j + 1) s_j \qquad \forall j \in J \tag{3.45}$$

$$P_k + 2 w_j n_t \geq p_j + n_t x_j \qquad \forall j \in J \tag{3.46}$$

$$P_k - 2(1 - w_j) n_t \leq p_j + n_t x_j - 1 \qquad \forall j \in J \tag{3.47}$$

Model 3.4.1: MIP model in batch-by-batch branch-and-bound

such that the remaining capacity, $b_r = b - \sum_j s_j x_j$, is less than the size s_j of the *smallest* job from the set of non-batched jobs that are *not longer* than the current batch. That is, if there exists an non-batched job j with $p_j \leq P_k$ and $s_j \leq b_r$, then the current assignment of jobs is infeasible in the model. The reasoning goes as follows: given any feasible schedule, assume there is a batch k_a with b_r remaining capacity and a later batch k_b containing a job j such that $s_j \leq b_r$ and $p_j \leq P_{k_a}$. Then

job j can always be moved to batch k_a without negatively affecting the quality of the solution: if the schedule was optimal, then moving j will not affect $L_{\max}$ at all (otherwise, it was no optimal schedule); if the schedule was not optimal, then $L_{\max}$ will stay constant (unless j was the longest job in k_b and $L_{\max}$ occured in or in a batch after k_b, in which case $L_{\max}$ will be improved).

This rule is implemented by means of a binary variable w_j, which, as defined by constraints (3.45) and (3.46), is 1 iff $p_j > P_k \vee x_j = 1$. These are the cases in which a job j is *not* to be considered in (3.44), and so w_j is used to scale s_j to a value insignificantly large in the eyes of the constraint's less-than relation.

After a solution is found, a child node in the search tree has run the subtree and returned, a constraint of the form

$$\sum_j x_j + \sum_i (1 - x_i) \leq n_j - 1 \quad \forall\{j \in J | x_j = 1\}, \forall\{i \in J | x_i = 0\} \tag{3.48}$$

is added to the model before the solver is called again, to exclude the last solution from the set of feasible solutions.

I still can't think of anything tighter ...

Using CP and cumulative packing after the batch

This approach is equivalent, but we now use CP to select the batch assignments, again based on a minimized $L_{\max}$ among the non-batched jobs. Model 3.4.2 uses interval variables j to represent the jobs; time constraints on the jobs ("est", "lft" etc.) are represented as equations involving the terms `startOf`(j) and `endOf`(j).

Contraints (3.50) through (3.53) bi-directionally define the relationship between a job's x_j and `startOf`(j): $x_j = 1$ is equivalent with a start time of $t = 0$, and $x_j = 0$ is equivalent with a start time of $t \geq P_k$. The term $\sum_{i \in J} p_i$ is used as a large constant as it is greater than any job's start time.

Constraint (3.57) is equivalent to constraints (3.45) through (3.47) in Model 3.4.1 above. The cumulative constraint (3.60) ensures that jobs do not overlap.

$$\text{Min.} \quad L_{\max} \tag{3.49}$$

$$\text{s.t.} \quad \texttt{startOf}(j) \geq (1 - x_j) P_k \qquad \forall j \in J \tag{3.50}$$

$$\texttt{startOf}(j) \leq (1 - x_j) \sum_{i \in J} p_i \qquad \forall j \in J \tag{3.51}$$

$$x_j \geq \frac{\frac{1}{2} - \texttt{startOf}(j)}{\sum_{i \in J} p_i} \qquad \forall j \in J \tag{3.52}$$

$$x_j \leq 1 + \frac{\frac{1}{2} - \texttt{startOf}(j)}{\sum_{i \in J} p_i} \qquad \forall j \in J \tag{3.53}$$

$$P_k \geq \max_{j \in J}(x_j p_j) \tag{3.54}$$

$$\texttt{endOf}(j) = d_j + L_{\max,\text{incmb}} - 1 \qquad \forall j \in J \tag{3.55}$$

$$L_{\max} \geq \max_{j \in J}(\texttt{endOf}(j) - d_j) \tag{3.56}$$

$$\texttt{IfThen}(p_j \leq P_k \wedge x_j = 0, b - \sum_{j \in J} s_j x_j \leq s_j) \qquad \forall j \in J \tag{3.57}$$

$$P_k + \frac{1}{b} \sum_i s_i p_i \leq d_j + L_{\max,\text{incmb}} - 1 - v_k \qquad \forall j \in J, \forall \{i \in J | d_i \leq d_j\} \tag{3.58}$$

$$\sum_j x_j \geq 1 \qquad \forall \{j \in J | d_j = \min(d_j)\} \tag{3.59}$$

$$\texttt{cumul}(J, b) \tag{3.60}$$

Model 3.4.2: CP model in batch-by-batch branch-and-bound

Results

4.1 Empirical comparison of described models

The models were tested on a set of job lists. Malapert's paper uses benchmark job lists by Daste et al. [?, ?]. Since neither publication is available online, I created my own set of randomized job lists for the purposes of this paper, with $s_j, p_j \in [1, 20]$ and $d_j \in [1, 10n_j]$ where n_j is the number of jobs. Ten different sample job sets are used per unique value of n_j. The times shown in figure 4.1 are averaged over those ten instances for each n_j.

The models were run on an i7 Q740 CPU in single-thread mode, with 8 GB RAM. Solving was aborted after a time of 300 seconds (5 minutes).

I have not yet measured "solution quality" in any way — I suppose the number of batches required would be a good metric?

The CP branch-and-bound model times out on most instances and is not shown here. The CP model times out on one 12-job instance and gets progressively worse with more jobs; similar to Malapert's original MIP model.

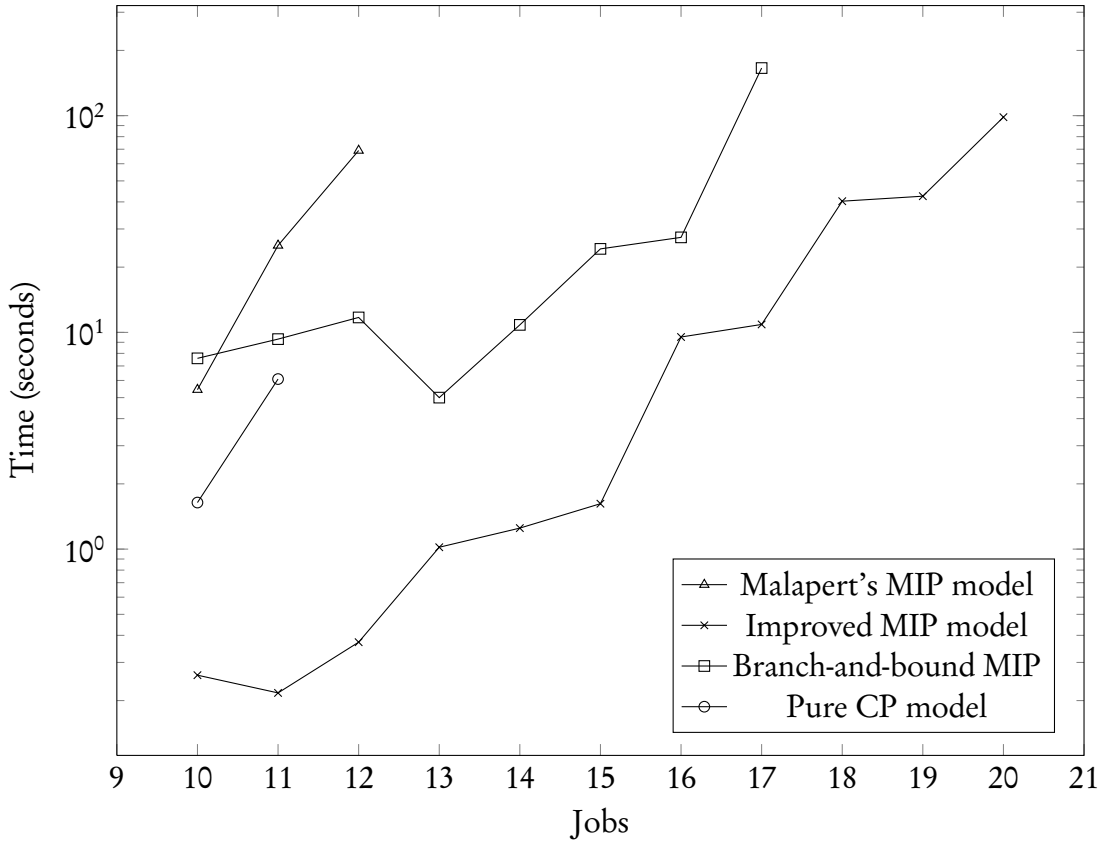


Figure 4.1: Comparison of CPU time used by different models to find an optimal schedule and prove its optimality.

Discussion

The improved MIP model currently performs best, mostly owed to constraint (3.20) which excludes a large number of potential batch assignments.

Unexplored ideas

6.1 MIP model

6.1.1 Upper bound on $L_{\max}$

An upper bound on $L_{\max}$ can be found by using a dispatch rule to find a feasible, if not optimal, schedule. A good approach could be the "best-fit" heuristic proposed in Malapert's paper.

6.1.2 Bounding the number of batches n_k

Initially, the number of batches needed is assumed to be equal to the number of jobs: $n_k = n_j$. Reducing n_k by pre-computing the maximum number of batches needed shrinks the x_{jk} matrix, and prunes potential search branches in branch-and-bound decomposition approaches.

Unfortunately, we cannot make a general statement that optimal solutions never have more batches than other feasible solutions – a simple counterexample is shown in figure 6.1.[1]

[1] To be more precise, we cannot state that at least one optimal solution is in the subset of feasible solutions that uses the fewest number of batches – a dominance situation that could be exploited, were it true.

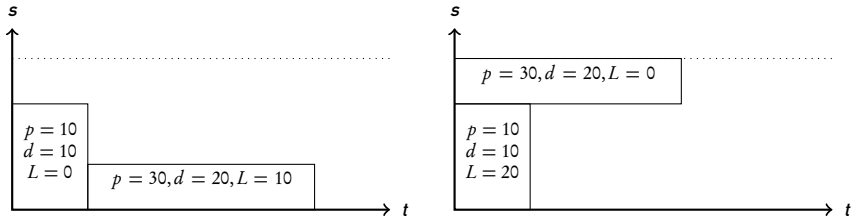


Figure 6.1: Overzealous batch elimination can increase $L_{\max}$

Starting out with a one-job-per-batch schedule sorted by EDD, we can explore all feasible batch configurations recursively. To generate any other feasible schedule (including the optimal solution), jobs j are rescheduled ("moved back") from their original batch k_{origin} into a prior batch k_{earlier}. This eliminates k_{origin} and requires, of course, that k_{earlier} has sufficient capacity. If the job's processing time p_j exceeds that of k_{earlier}, then the lateness of batches between k_{earlier} and k_{origin} will increase; the merit of such a move cannot be judged a priori, so it is an *unsafe* batch elimination. *Safe* eliminations, on the other hand, will never worsen $L_{\max}$, and only they can be considered when bounding n_k a priori.

Algorithm 2 outlines a recursive method to find an upper bound on n_k, recognizing safe batch eliminations only.

Algorithm 2 Recursive algorithm to find an upper bound on n_k

```
if no open jobs left then                          ▷ if this is a "leaf node" in the recursion
    update UB(n_k)
end if
find the combination of unsafe later jobs that fills up the capacity most, leaving us with capacity b_r
find all combinations x of safe later jobs that fit into b_r
repeat
    x = next safe job combination
    ignoreJobs ← x                                 ▷ make the moves, let the next recursion level deal with the rest of the jobs
    spawn and run child node with J \ ignoreJobs
    ignoreJobs ← ignoreJobs \ x
until all combinations have been explored
return
```

This algorithm evidently requires exponential time. A relaxed variant is a possible option: if only a single unsafe move *into* a batch is possible, no safe eliminations into that batch are considered at all and we skip to the next batch. This would greatly speed up the recursion but also significantly weaken the usefulness of the resulting upper bound.

In a brach-and-bound decomposition approach in which batches are modelled individually, an upper bound on n_k could be used to limit the depth of the search tree, or, in combination with a method to determine a lower bound on the remaining jobs' n_k at every node, to actively prune the search tree during the search. The latter method, however, would also run in exponential time as it, again, would require knapsack-type reasoning unless we use a much less powerful relaxation.

6.2 CP Model

6.2.1 Constraint on number of batches with length $P_k > p$

Since batches take on the processing time of their longest job, there is at least one batch with $P = \max_j p_j$. We can proceed to fill batches with jobs, ordered by non-increasing processing time, based on algorithm 3.

I haven't found an elegant way to turn this into a gcc constraint, but I also fail to see how it would be useful – this constraint matches all feasible solutions, not just a superset of the optimal one.

Algorithm 3 Generating lower bounds on batch lengths

```
J* ← J                                          ▷ initialize all jobs as unassigned jobs
n_k ← 1; S_k ← {0}; P_{k,min} ← {0}             ▷ Create one empty batch of size and length zero
sort J* by processing time, non-increasing
repeat
    j ← J*.pop()                                ▷ select job for assignment, longest job first
    loop through all n_k existing batches k, first batch first
        k_p ← ∅                                 ▷ no feasible batch
        c_min = b                               ▷ currently known minimum remaining capacity
        if s_j < b − S_k and b − S_k < c_min then
            k_p ← k_p; c_min ← b − S_k
        end if
    end loop
    if |k_p| = 1 then
        S_{k_p} ← S_{k_p} + s_j                 ▷ assign job j to batch k_p
    else
        if n_k < LB(n_k) then
            n_k ← n_k + 1                       ▷ open new batch
            S_{n_k} ← s_j; P_{n_k,min} ← p_j    ▷ assign s_j and p_j to the new batch
        else
            leave the loop now and end.
        end if
    end if
until J* is empty
```

At the end of this algorithm, we can state:

$$\texttt{globalCardinality}(|P_{k-1,\min} > P_k \geq P_{k,\min}| \geq 1) \quad \forall k \in \{k_0, \ldots, k_{LB(n_k)}\} \tag{6.1}$$

The algorithm sorts jobs by non-increasing p, and then fills batches job by job. If a job fits into a previous batch, it is assigned there. If a job fits into multiple previous batches, it is assigned to the batch with the smallest remaining capacity. This is called *best-fit decreasing* rule, and works as follows: let $J^\star$ be the set of jobs sorted by p, then at least one batch will be as long as the longest job $j_1^\star$. If the next n jobs fit into this batch, then there is at least one batch not shorter than $j_{n+1}^\star$, and

similarly for subsequent batches.

Unfortunately, the optimal solution may perform better than the packing heuristic in terms of "vertical" (s_j) bin packing, and may thus require fewer batches. We therefore need to find a lower bound $LB(n_k)$ on the number of batches, and we can only guarantee the first $LB(n_k)$ of the above constraints to hold in the optimal solution. Finding a true lower bound is a two-dimensional bin packing problem, which runs in exponential time, so we have to come up with an even lower bound – right now I can only think of j_0, the number of jobs ordered by decreasing s_j that can never fit into a batch together.

6.2.2 Constraint on number of batches with length $D_k > d$

In a similar fashion, we can determine that the second batch must be due no later than the earliest-due job j_{m+1} that can *not* fit into the first batch – if we sort jobs by due date and fit the earliest m jobs into the first batch – and so on for subsequent batches. Once again, since best-fit decreasing may not perform optimally in terms of "vertical" packing, this may not be valid for batches beyond the known $LB(n_k)$.

6.2.3 All-different constraints on P_k and D_k

I haven't found a way to implement this efficiently in CP Optimizer; I also don't quite see how it would constrain anything.

Furthermore, if all jobs have different processing times, all batches will have different processing times as well: `alldifferent`(P_k). If m out of n_j jobs have different processing times, we can still enforce `k_alldifferent`(P_k, m).

Similarly, we knows that the constraint `k_alldifferent`(D_k, m) must be true if m out of n_j jobs have different due dates.

6.3 Decompositions

6.3.1 Potential heuristics

Improve the initial $L_{\text{max,incumbent}}$ A better initial upper bound on $L_{\max}$ can help prune some branches of the search tree from the outset. There are several dispatch rules (or maybe other heuristics?) that could be explored to do this better.

Improve $L_{\text{max,incumbent}}$ during search It may be useful to use a heuristic like above to "complete the schedule" once a promising partial schedule has been generated. I have yet to identify situations where this is always helpful.

6.3.2 Move-back decomposition

Another possible way to set up a branch-and-bound search for solutions works as follows: first, consider all jobs to be "single", i.e. assigned to individual batches such that $B_j = k_j \ \forall j \in J$. Compute the $L_{\max}$ for this schedule. Then, at every level of the search tree, move some single job j into any earlier batch $k \leq k_j$, but only if that move does not violate k's capacity.

If we start with a schedule of single jobs and only allow moving single jobs into earlier batches (and any schedule can be produced by a sequence of such moves!), we maintain the EDD ordering of batches. More importantly, such moves will never shift the position of $L_{\max}$ to the right:

In any partial schedule following EDD, let k be the batch with maximum lateness $L_k = L_{\max}$. It has processing time p_k. Then the lateness of the batch before k must be $L_{k-1} \geq L_k - p_k$ as a consequence of the EDD sequencing. Any batch following k can have a lateness no greater than $L_k - 1$.

- Moving any single job from a batch following k into a batch before k will worsen $L_{\max}$, but not change its position. Such a move is never necessary to arrive at an optimal solution.
- Moving back any single job from a batch before k *safely*[2] will improve $L_{\max}$, but not change its position.
- Moving back a single pre-k job j from a batch β into an earlier batch α *unsafely* will reduce L_k by p_α; $L_{\max}$ may still be in k, or it may be found in any batch between (and including) α and β, since their lateness $L_{[\alpha,\ldots,\beta]}$ is now increased by $p_j - p_\alpha$.
- If the max-lateness job j is single itself, it can be moved from k into an earlier batch α. If this is done *safely*, the batch immediately preceding k will still have a lateness $L_{k-1} \geq L_k - p_j$. All batches after k will have their lateness reduced by p_j, but since their maximum lateness did not exceed $L_k - 1$ before the move, it will now be at least 1 less than that of batch $k - 1$.
- If the max-lateness job j was moved back *unsafely* into a batch α, batches between and including α and $k - 1$ now have their lateness increased by $p_j - p_\alpha$, while batches after k have their lateness decreased by p_α. Again, batch L_{k-1} would exceed the maximum lateness of batches after k.

[2] for the meaning of "safe" and "unsafe" moves, compare section 6.1.2.

This shows that after a sequence of operations in which single jobs are moved into earlier batches, $L_{\max}$ will never shift to the right. In fact, this means that all jobs after the $L_{\max}$-job in a single-job EDD schedule can be ignored in the scheduling problem entirely, although this turns out to be quite inconsequential as that job is often at or near the end of the single-job EDD schedule, resulting from the fact that $L_{k-1} \geq L_k - p_k$. By the same token, high-quality solutions often have their $L_{\max}$ in an early batch. This may give rise to some sort of decomposition method in which jobs are strategically moved back, and where we are trying to move $L_{\max}$ to the left as much as possible.